To Home Page of EPP. To Home Page of Y.Ichisugi.

English documents of EPP (Preliminary version)

1999-03-02

CAUTION: Some of these documents are not proofed by Ichisugi. There must be a lot of mistranslation which are technically incorrect.

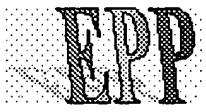
epp-slides19990117.pdf(582KB)
Slides describing reseach purpose and overview of EPP.

epp-parser.pdf(233KB)
Yuuji ICHISUGI: Modular and Extensible Parser Implementation using Mixins(DRAFT)
(Japanese version of this paper is in Information Processing Society of Japan, Transaction on Programming, Vol.39 No.SIG 1(PRO 1), pp.61--69, December, 1998.)

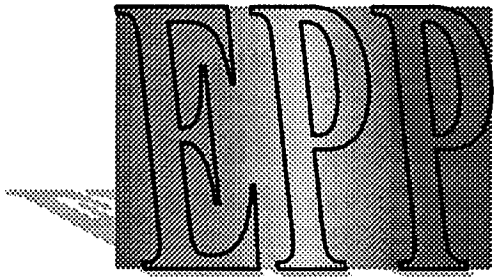
epp-type-check.pdf(198KB)
Yuuji ICHISUGI: Extensible Type System Framework for a Java Pre-Processor : EPP (DRAFT)
(Japanese version of this paper will appear in proceedings of SPA'99.)

EPP documents for plug-in programmers (DRAFT) [edocalpha.zip(177KB)]
Detailed documents including API document generated by javadoc.

To Home Page of EPP. To Home Page of Y.Ichisugi.



Extensible Java Pre-processor



Electrotechnical Laboratory

Yuuji Ichisugi

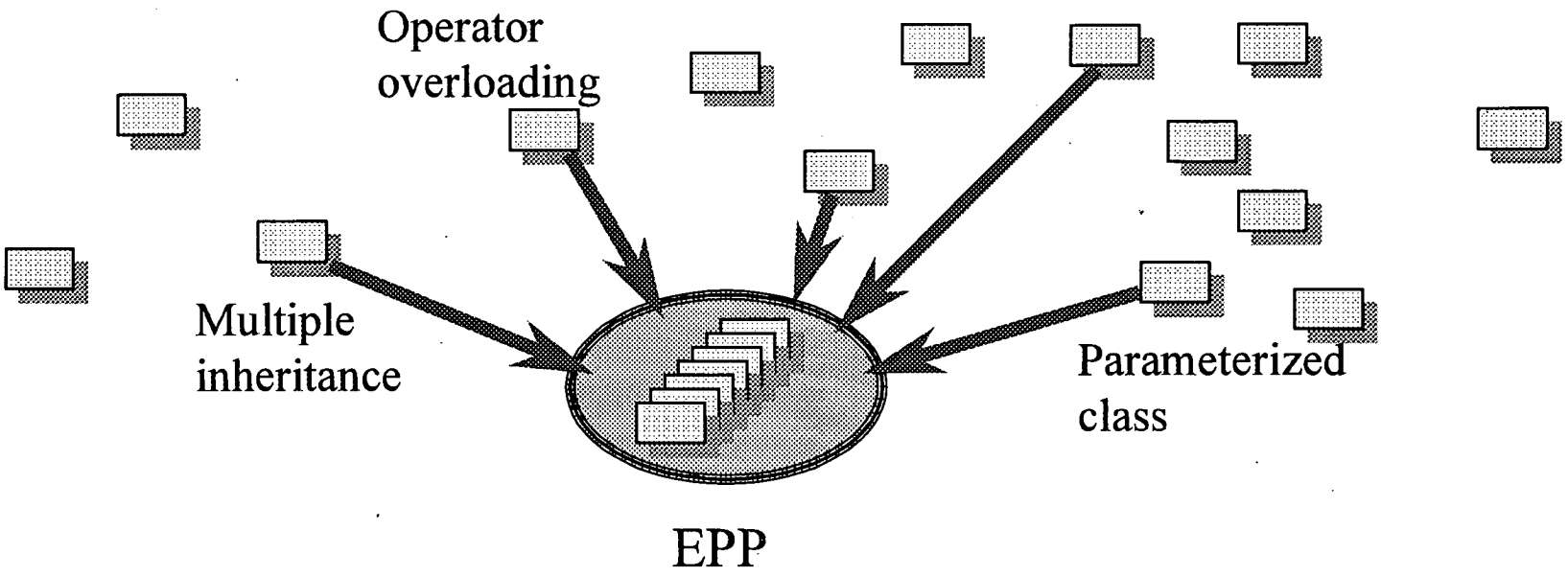
http://www.etl.go.jp/~epp/

Contents

- What is EPP ?
- Examples
- Mixin
- Extensible recursive-descent parser
- Extensible type checking system
- Conclusion

What is EPP ?

- A pre-processor which can be extended by adding "plug-ins".



World-wide programming language

- EPP will produce various programming languages required for world-wide programming.
- Example: **Mobile agent system**
  - Collaboration with Tsukuba University.
  - Thread migration implementation on pure JavaVM.
- Future plan: **Security enhancement** for Java
  - taint Java, proof-carrying code for Java, …?

Example : Symbol plug-in

EPP Plug-in

```
#epp jp.go.etl.epp.Symbol

import  jp.go.etl.epp.epp.Symbol;

public class TestSymbol {
  public static void main(String args[]){
    Symbol x = :foo;
    Symbol y = :"+";
    System.out.println(x == :foo); // true
    System.out.println(y == :foo); // false
  }
}
```

Translated program

```
/* Generated by EPP 1.0.2beta (by lisp-epp1.0.2beta) */
import jp.go.etl.epp.epp.Symbol;
public class TestSymbol {
  private static final Symbol _Sym0 = Symbol.intern("foo");
  private static final Symbol _Sym1 = Symbol.intern("+");
  private static final Symbol _Sym2 = Symbol.intern("foo");
  private static final Symbol _Sym3 = Symbol.intern("foo");
  public static void main(String args[]){
    Symbol x = _Sym0;
    Symbol y = _Sym1;
    System.out.println((x) == (_Sym2));
    System.out.println((y) == (_Sym3));
    }
  }
```

Actually, all line numbers are preserved by translation in order to support debugging.

Source code of Symbol plug-in

```
#epp jp.go.etl.epp.Symbol
#epp jp.go.etl.epp.SystemMixin
#epp jp.go.etl.epp.AutoSplitFiles
#epp jp.go.etl.epp.BackQuote
#epp jp.go.etl.epp.EppMacros

package jp.go.etl.epp.epp.Symbol;

defineNonTerminal(symbol, symbolOtherwise());
SystemMixin SymbolRep {
  class Epp {
    extend Tree primaryTop() {
      if (lookahead() == ':') {
        return symbol();
      } else {
        return original();
      }
    }
    extend Tree symbolTop() {
      if (lookahead() == ':') {
        matchAny();
        Token next = matchAny();
        if (next.isSymbol()) {
          return new Tree(:symbol, new Identifier((Symbol)next));
        } else if (next.isLiteralToken()) {
           String contents = next.literalContents();
          return new Tree(:symbol, new Identifier(Symbol.intern(contents)));
        } else {
          throw error("Symbol plug-in: "+ next+ "appeared after ¥":¥" .");
        }
      } else {
        return original();
      }
    }
    define implement Tree symbolOtherwise() {
      throw error("symbol is required here.");
    }
    extend void initMacroTable() {
      original();
      defineMacro(:symbol, new SymbolMacro());
    }
  }
}
class SymbolMacro extends Macro {
  public Tree call(Tree tree){
    checkArgsLength(tree, 1);
    Tree[] args = tree.args();
    String str = args[0].idName().toString();
    Tree var = new Identifier(genTemp("_Sym"));

    addTree(:beginningOfClassBody,
            (decl (modifiers (id private) (id static)
                             (id final))
              (id Symbol)
              (vardecls
               (varInit ,var
                (invokeLong (id Symbol) (id intern)
                 (argumentList ,(new LiteralTree(:string, str))))))));
    return var;
  }
}
```

Grammar extension

Macro expansion

Grammar extension

```
#epp jp.go.etl.epp.Symbol
#epp jp.go.etl.epp.SystemMixin
#epp jp.go.etl.epp.AutoSplitFiles
#epp jp.go.etl.epp.BackQuote
#epp jp.go.etl.epp.EppMacros

...

   extend Tree symbolTop() {
      if (lookahead() == :":") {
        matchAny();
        Token next = matchAny();
        if (next.isSymbol()){
          return new Tree(:symbol, new Identifier((Symbol)next));
        } else if (next.isLiteralToken()){
          String contents = next.literalContents();
          return new Tree(:symbol,
                          new Identifier(Symbol.intern(contents)));
        } else {
          throw Epp.fatal("");
        }
      } else {
        return original();
      }
    }
```

Macro expansion

```
class SymbolMacro extends Macro {
  public Tree call(Tree tree){
    checkArgsLength(tree, 1);
    Tree[] args = tree.args();
    String str = args[0].idName().toString();
    Tree var = new Identifier(genTemp("_Sym"));

    addTree(:beginningOfClassBody,
            `(decl (modifiers (id private) (id static)
                      (id final))
               (id Symbol)
               (vardecls
                (varInit ,var
                 (invokeLong (id Symbol) (id intern)
                  (argumentList ,(new LiteralTree(:string, str))))))));
    return var;
  }
}
```

GJ(Generic Java)

```
#epp jp.go.etl.epp.Generic
import java.util.Stack;

public class TStack<T> {
  private Stack stack = new Stack();
  void foo(){
    T x = null;
  }
  void push(T val){
    stack.push(val);
  }
  T pop(){
    return (T)stack.pop();
  }
}
```

Multiple inheritance

```
#epp jp.go.etl.epp.MultipleInheritance

class C1 {
  int m11() { return 11; }
  int m12() { return 12; }
}
class C2 {
  int m21() { return 21; }
  int m22() { return 22; }
}
class C3 {
  int m31() { return 31; }
  int m32() { return 32; }
}
class C4 extends C1, C2, C3 {
  int m41() { return 41; }
  int m42() { return 42; }
}
```

Composability

- Generic and Multiple Inheritance plug-ins.

```
#epp jp.go.etl.epp.Generic
#epp jp.go.etl.epp.MultipleInheritance

import java.util.Stack;
class TStack<T> extends A1, A2 {
  private Stack stack = new Stack();
  void foo(){
    T x = null;
  }
  void push(T val){
    stack.push(val);
  }
  T pop(){
    return (T)stack.pop();
  }
}
```

Plug-ins currently implemented

- C, C++, g++ features
  - #ifdef, #include, enum, defmacro, assert macro, operator overloading, optional parameters, typeof
- Lisp features
  - Symbol, back quote macro, mixin, progn
- ML features
  - parameterized modules
- Perl features
  - association array

Design goal of EPP

- **High extensibility**.
  All extensions which can be done by "editing of original source-code" should be able to be done by EPP plug-ins.
- **High composability.**
  Plug-ins should work correctly even if they are used with other plug-ins.
- (C.f. Language extension systems based on attribute grammar or term rewriting system…)

EPP can be used as framework of ...

- Java extensions.
  - Operator overloading, parameterized modules, ...
- Source-code analyzing tools.
  - Metrics, cross reference, ...
- Source-code translation tools.
  - source level optimization, obfuscation, ...
- **Aim: A standard platform of Java related tools.**

Why EPP is useful as framework?

- **EPP will support all the following troublesome work.**
- Java grammar parser (Cf. JavaCC, ANTLR,...)
- Abstract syntax tree manipulation libraries.
- **Type checking**.

Now implementing the followings.

- Automatic re-compilation.
- ...

Aim: Making a programming language like a "**PC compatible machine**"

- EPP provides **the standard interface** for extension modules for language processors.
- Small venture companies or Universities can produce practical language parts.
- **Free competition**.
- EPP will accelerate the progress of programming languages.

For language researchers

- Before EPP (PhD languages)
  - More than half an year to implement an experimental programming language system.
  - Developing tools(debuggers, etc.) are future work.
  - Only a few users.
- After EPP
  - 3 hours ~ 3 month to implement.
  - Existing developing tools can be reused.
  - 100~ 10000 users can try the new language.
  - The idea becomes practical system immediately.

For programmers

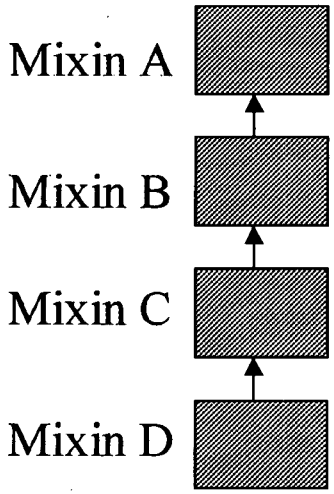
- Before EPP
  - Language features **selected by the designer** to suit his own tastes are imposed.
    (No free competition !)
  - The latest research results are not adopted to the user's language processor.
- After EPP
  - Programmers can select their **favorite** features.
  - The latest research results can be used immediately.

Technical characteristics

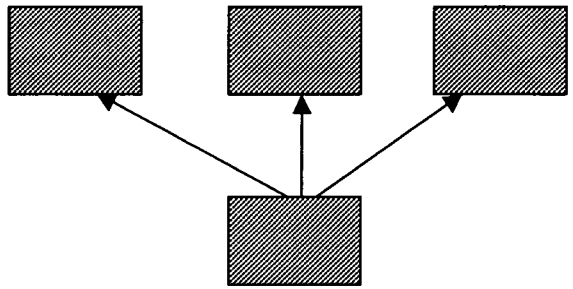
- **Mixin based design** which increase extensibility and re-usability of modules
  - Traditional object-oriented design is not enough.
  - Design pattern is still not enough.
- **Extensible architecture**
  - extensible lexical analyzer
  - extensible recursive descent parser
  - extensible type checking system

What is mixin ?

- "Abstract subclasses" which are not depending on specific super classes.
- A class can be constructed by composing and linearizing mixins.



NOTE: It is impossible to implement using C++ multiple inheritance.



Why mixins are important in EPP?

- Mixins supports programming-by-difference and modular programming.
  - Mixins increases composability of EPP plug-ins.
- Mixin mechanism is provided as an EPP plug-in. (Mixins can be separately compiled.)
- EPP itself and EPP plug-ins are written in Java with the "Mixin plug-in."
  - And bootstraped by EPP written in Common Lisp.

A program NOT using mixins.

- Not flexible.

```
class Foo {
  void m(char c){
    if (c == 'B') {
      doB();
    } else if (c == 'A') {
      doA();
    } else {
      doDefault();
    }
  }
}
```

A program using mixins.

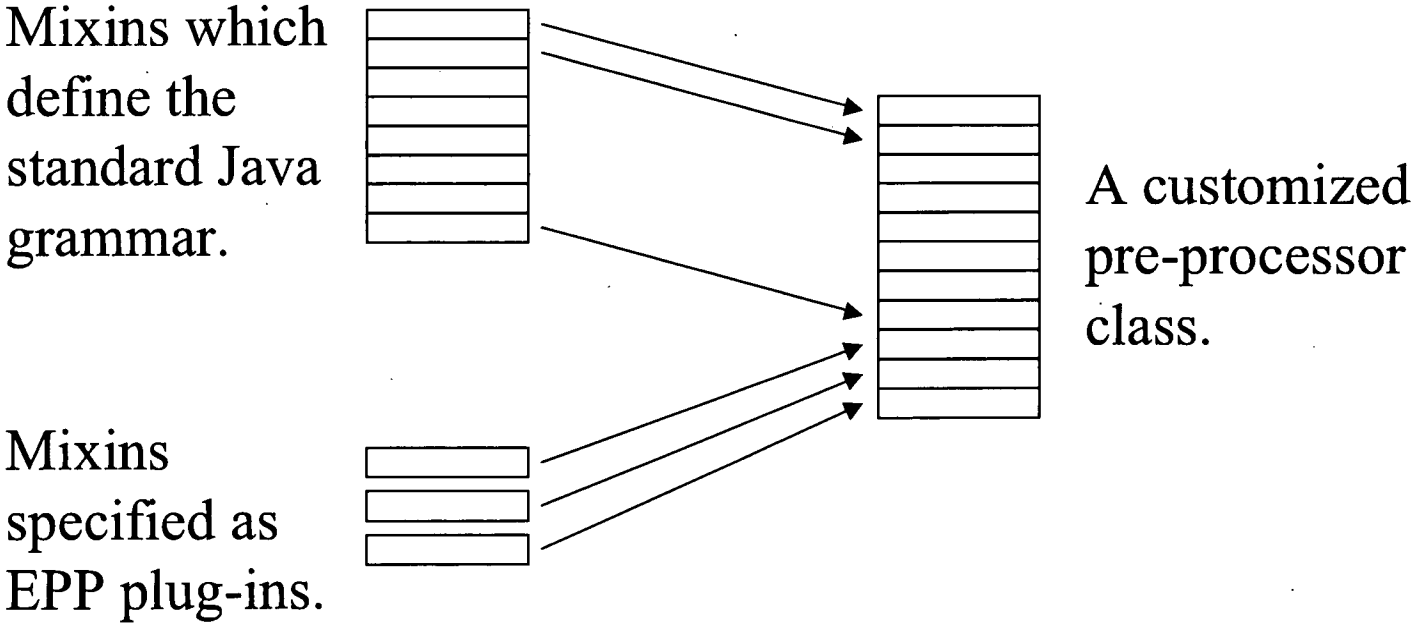
```
SystemMixin Skeleton {  class Foo {
   define void m(char c){ doDefault();   }
}}
```

```
SystemMixin A {  class Foo {
   void m(char c){ if (c == 'A') { doA();} else { original(c); }}
}}
```

```
SystemMixin B {  class Foo {
   void m(char c){ if (c == 'B') { doB();} else { original(c); }}
}}
```

How EPP starts pre-processing?

- The EPP main routine composes mixins, constructs one pre-processor class, and executes it.



How to write parser with modular and extensible way.

- Recursive descent parser consists of nested if statements.
- Nested if statements can be written with modular way using mixins.
- Each non-terminal and each alternative becomes a mixin.

An example of a non-terminal.

Exp → ++ Exp | (Exp) | Term += Exp | Term
| Exp + Term | Exp ++

This can be rewritten and become LL(1).

Exp → ExpTop ExpLoop
ExpTop → ++ Exp | (Exp) | Term ExpRight
ExpRight → += Exp | e
ExpLoop → ExpLeft ExpLoop | e
ExpLeft → + Term | ++

Recursive Descent Parser(without mixins)

```
Tree exp() {
 Tree tree = expTop();
 while (true){
  Tree newTree = expLeft(tree);
  if (newTree == null) break;
  tree = newTree;
 }
 return tree;
}

Tree expTop() {
 if (lookahead() == :"++"){
  matchAny();
  return new Tree(:"preInc", exp());
 } else if (lookahead() == :"("){
  matchAny();
  Tree e = exp();
  match(:")");
  return new Tree(:"paren", e);
 } else {
  return expRight(exp1());
 }
}
```

```
Tree expRight(Tree tree) {
 if (lookahead() == :"+="){
  matchAny();
  return new Tree(:"+=", tree, exp());
 } else {
  return tree;
 }
}

Tree expLeft(Tree tree) {
 if (lookahead() == :"+"){
  matchAny();
  return new Tree(:"+", tree, exp1());
 } else if (lookahead() == :"++"){
  matchAny();
  return new Tree(:"postInc", tree);
 } else {
  return null;
 }
}

Tree exp1() { return term(); }
```

Extensible parser skeleton.

```
SystemMixin Exp {
  class Parser {
    define Tree exp(){
      Tree tree = expTop();
      Tree newTree;
      while (true){
        newTree = expLeft(tree);
        if (newTree == null) break;
        tree = newTree;
      }
      return tree;
    }
    define Tree expTop(){ return expRight(exp1()); }
    define Tree expRight(Tree tree) { return tree; }
    define Tree expLeft(Tree tree){ return null; }
    define Tree exp1() { return term(); }
  }
}
```

Exp → Term

Adding a left associative operator.

```
SystemMixin Plus {
  class Parser {
    Tree expLeft(Tree tree) {
      if (lookahead() == :"+") {
        matchAny();
        return new Tree(:"+", tree, exp1());
      } else {
        return original(tree);
      }
    }
  }
}
```

Exp → Exp + Term

Adding a right associative operator.

```
SystemMixin Assign {
  class Parser {
    Tree expRight(Tree tree) {
      if (lookahead() == :"+=") {
        matchAny();
        return new Tree(:"+=", tree, exp());
      } else {
        return original(tree);
      }
    }
  }
}
```

Exp → Term += Exp

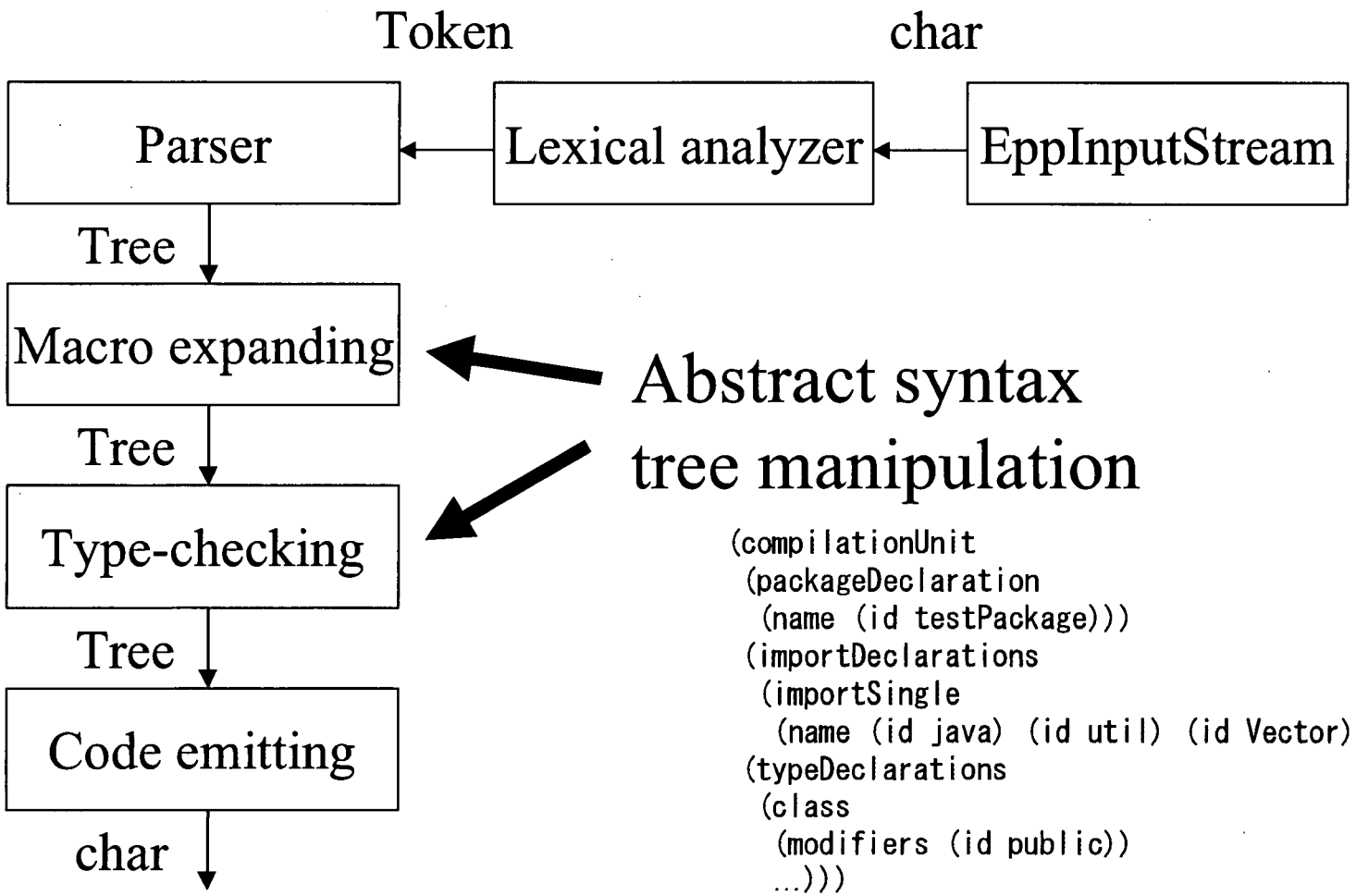
Java grammar

- 105 mixins.
  - 29 mixins define non-terminals and others define alternatives.
- I used 4 explicit backtrack because some production rules are not LL(1).

EPP architecture

- Simple.
  - Easy to understand its behavior.
- General-purpose.
  - Data structures (**Token**, **Tree** and **Type**) can express various programming languages.
- Extensible.
  - All parts of EPP are extensible.

4 passes

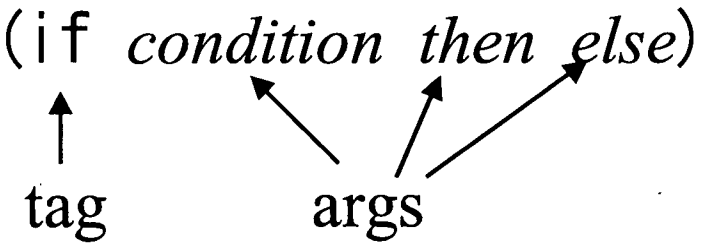


Abstract syntax tree manipulation

```
(compilationUnit
 (packageDeclaration
  (name (id testPackage)))
 (importDeclarations
  (importSingle
   (name (id java) (id util) (id Vector)))
 (typeDeclarations
  (class
   (modifiers (id public))
   ...)))
```

Abstract Syntax Tree

- Tree is an immutable object.
- Simple API:
  - Symbol tag();
  - Tree[] args();
  - Type type();
  - ...

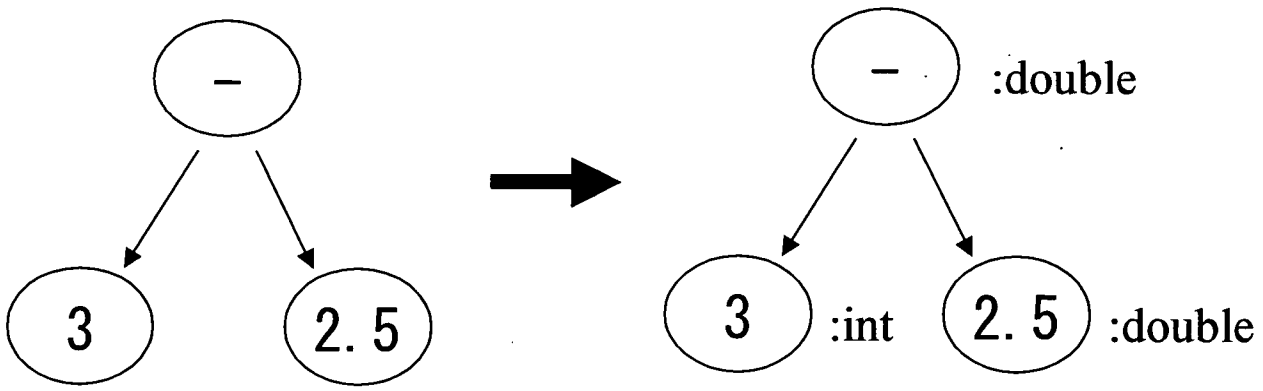


- "Back quote macro" plug-in is provided.

```
`(invokeExp ,target ,mehodName
            (argumentList ,@args))
```

Type checking pass

- Translate AST which does not have type information into AST which has.



Type checker object

- "TypeChecker" is associated with all tree tags.
- Extensible using *decorator pattern*.
  - Example: operator overloading

```
class ExpandOperatorOverloadingOfMinus extends TypeChecker {
  public Tree call(Tree tree) {
    Tree[] newArgs = typeCheckArgs(tree);
    Type t1 = newArgs[0].type();
    if (t1.tag() == :class){
      // Return the AST of "e1.minus(e2)" .
      return `(invokeExp ,(newArgs[0])
              (id minus) (argumentList ,(newArgs[1])));
    } else {
      return orig.call(tree.modifyArgs(newArgs));
    }
  }
}
```

Relationship between types

- Extensible using *mixin mechanism*.
  - **Not OO style.**
  - **But as modular and extensible as OO style.**
- API:
  - `boolean isSuperType(Type t1, Type t2);`
  - `Tree selectMostSpecificMethod(...);`
  - ...

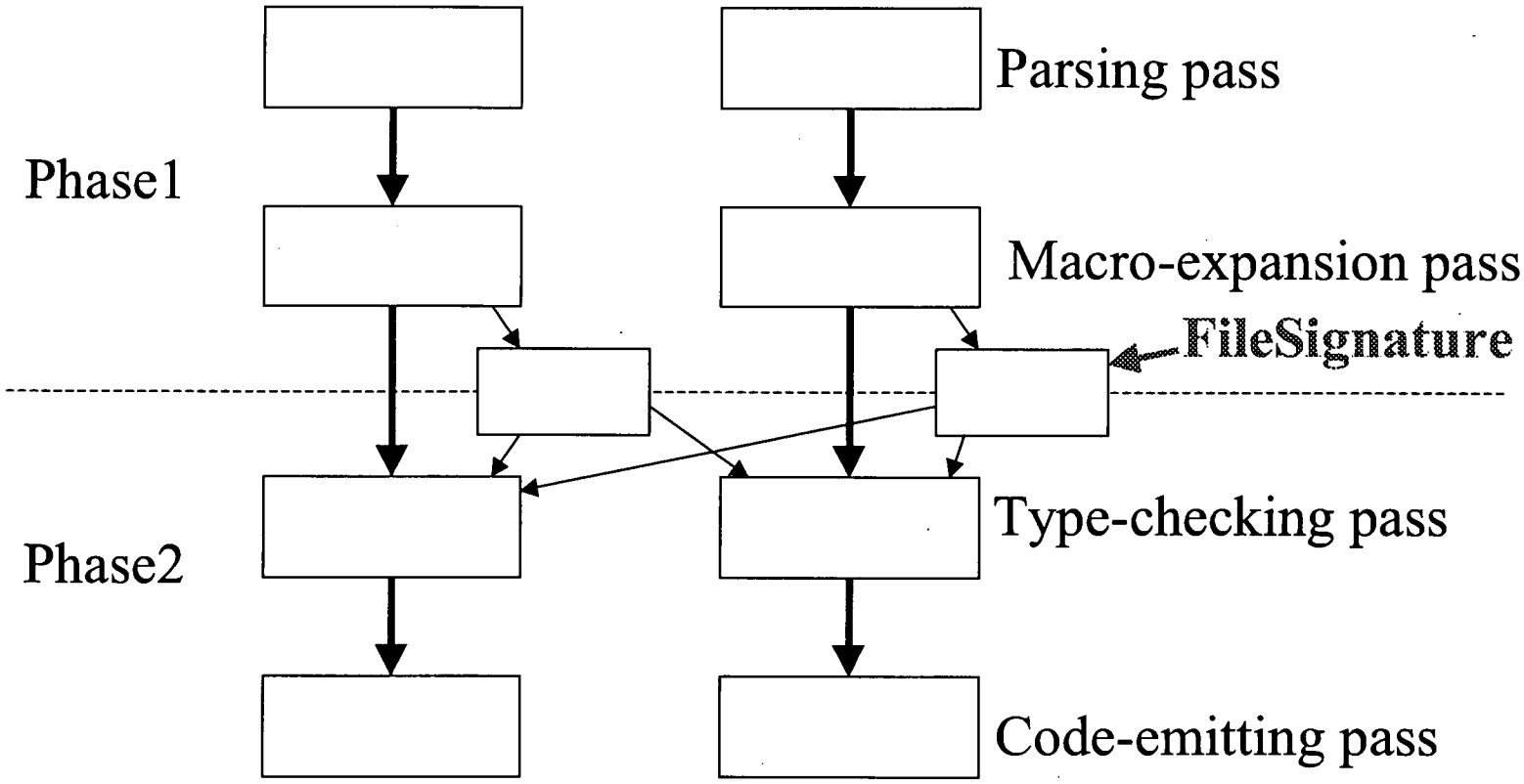
Example: extending isSuperType

```
boolean isSuperType(Type t1, Type t2){
    if (t1.tag() == :assocArray){
      if (t2.tag() == :assocArray){
        AssocArrayType a1 = (AssocArrayType)t1;
        AssocArrayType a2 = (AssocArrayType)t2;
        // Contra-variant role.
        return isSuperType(a1.getKeyType(),
                           a2.getKeyType())
           && isSuperType(a2.getValueType(),
                           a1.getValueType());
      } else {
        return false;
      }
    } else {
      return original(t1, t2);
    }
  }
```

Separate compilation

- Java language:
  - No forward declarations / header files.
  - No Makefiles.
  - These information is automatically generated by **javac** and written into class files.
- EPP's framework has generalized javac's mechanism.

Separate compilation of EPP



Files do not depend on each other directly.

Related work

- Nothing except for EPP achieve both of high extensibility and high composablity.
  - Parser Generator(Yacc,JavaCC,ANTLR, ...)
    Compiler Generator (Eli, Rie,...)
  - Extensible Language (EL1, lisp, ...)
  - Reflection (3Lisp, ABCL/R3,AL/1, ...)
  - Meta Object Protocol (CLOS-MOP)
  - Extensible translator (TXL, Sage++, Camlp4, ...)
  - Compile Time MOP (MPC++, OpenC++, OpenJava, JTRANS)

Application 1. Mobile agent system

- PLANET project at Tsukuba University
- Pure Java implementation of thread migration.
  - **No JavaVM modification**.
  - **Almost 0% overhead** will be achieved.(?)
- Context saving/restoring code is added.
- Related work:
  - [Funfrocken98] 10% overhead.
  - Porch [Ramkumar, Strumpen97] For C language. More than 100% overhead.

Application 2. Java metrics tool

- JavaMetrics project by Electrotechnical laboratory, el.al.
- Enhance reliability of Java source code.
  - Code audit.
  - Code metrics.
  - Instrumentation for test coverage.

Conclusion

- EPP plug-ins can extend the pre-processor with programming-by-difference style.
- EPP achieves both of high extensibility and high composability.
- Separate compilation is not implemented yet.
- Distributed with source-code and examples.
  - http://www.etl.go.jp/~epp/

Modular and Extensible Parser Implementation using Mixins (DRAFT)

Yuuji ICHISUGI
Electrotechnical Laboratory

March 1, 1999

Abstract

This paper describes a method to construct highly modular and extensible recursive descent parser. This parser is used in an extensible Java preprocessor, EPP. EPP can be extended by adding plug-ins which extend Java syntax and add new language features. The EPP's parser consists of small mixins. A recursive descent parser class is constructed by composing these mixins. The syntax accepted by the parser can be extended by adding new mixins.

1 Introduction

The author has developed an extensible preprocessor for Java[GJS96], EPP[IR97, Ich] which can be extended by adding new modules which extend Java syntax, and added new language features. This paper describes a method of constructing a highly modular and extensible recursive descent parser, which is the parser of EPP.

In designing EPP, we aimed at a wide-range of extensibility, highly flexible implementation of extension modules, and simultaneous usability of multiple extension features - hereafter called composability.

Traditionally, extension of a system requires editing and direct modification of the source code. This has been considered the best method of realizing the highest extensibility and flexibility in implementation. With this method, however, it is difficult to simultaneously use more than one independent extension: the user cannot achieve composability.

Many extensible languages define a new syntax and its semantics with a declarative description. While this realizes high composability, descriptiveness of a descriptive language limits extensibility and flexibility.

EPP realizes as high extensibility as editing source code and realizes as high composability as declarative description. EPP's parser was implemented in such a way as to treat its extension modules as ordinary general-purpose programming language modules, using mixins, the feature of programming by difference. The parser has the following features:

Plug-ins which define new syntax, operators and others can be added afterward. Because syntax extension plug-ins are programmed by difference, multiple plug-ins can be combined at the same time.

Because of high modularity, plug-ins can be separately compiled one by one. Syntax extension plug-ins can be distributed without source codes.

Because of its general-purpose language description, it is easy to perform such ad-hoc processes as context sensibility and global escape, which are rather awkward with the BNF method.

Error recovery and line number managing can be implemented.

The EPP's architecture is applicable to any other programming languages if the implementation language has symbol and mixin features.

This paper consists of seven more sections. Section2 outlines EPP, Section3 presents symbols and mixins-language features necessary in parser implementation, Sections4 and 5 explain how to describe the extensible parser using mixins and to implement ad-hoc processes, Section6 evaluates the parser implementation, Section7 describes related studies, and Section8 is the conclusion.

2 Outline of Extensible Java Pre-processor EPP

EPP is an extensible Java source-to-source preprocessor which can introduce new language features. The user can specify EPP plug-ins at the top of the Java source code by writing `#epp name` in order to incorporate various extensions of Java. Multiple plug-ins can be retrieved simultaneously as

long as they do not collide with each other. Emitted source codes can be compiled by ordinary Java compilers and debugged by ordinary Java debuggers.

EPP works not only as an extensible Java preprocessor but also as a language-experimenting tool for language researchers; a framework for extensible Java implementation; and a framework for a Java source code parser/translator.

The EPP's source code is written in Java extended by EPP itself, and it was bootstrapped by EPP written in Common Lisp[Ste90]. The byte-code is available in any platform where Java is supported.

3 symbol and mixins

This section describes symbols and mixins, which are the language features necessary in implementing the extensible parser.

3.1 Symbol Implementation on Java

Symbols, a data-type as in languages like Lisp, have the following features:

Symbols are similar to constants defined by C language's **enum** statement. Unlike C constants, however, the user can use symbols having arbitrary names (strings) as required in the source codes, without defining a finite number of elements beforehand.

Symbols are similar to string literals, but unlike strings, the mere pointer comparison efficiently judges equality of symbols.

By specifying a name, a symbol can be generated dynamically.

Fig.1 shows a program using the Symbol plug-in. A Symbol literal is expressed by a colons followed by an identifier or a string literal.

The Symbol plug-in implements symbols on Java as follows: The symbol literals in the program are translated into references to each individual private static final variable. The variable is initialized by the return value of a static method invocation : `Symbol.intern("name")`. `Symbol.Intern("name")` looks up a hashtable and returns an instance of Symbol class having the specified name if it already exists. If it does not, a new instance is generated, registered in the table and returned. As a result, symbol literals having the same name are guaranteed to reference the instances having the same identity. The execution does not produce the overhead of hashtable retrieval because only a static constant value is referenced.

```
#epp jp.go.etl.epp.Symbol
import jp.go.etl.epp.epp.Symbol;

public class TestSymbol {
  public static void main(String args[]){
    Symbol x = :foo;
    Symbol y = :"+";
    System.out.println(x == :foo); // true
    System.out.println(y == :foo); // false
  }
}
```

Figure 1: A program using Symbol plug-in

3.2 Mixin Implementation on Java

3.2.1 What is mixins?

Usually, in an object-oriented language, a particular super class name is specified when defining a subclass; a mixin is a subclass defined with no particular super class specified. A mixin is defined on the presumption that multiple inheritance and linearization by another class will determine a super class afterward.

Bracha[BC90] showed that the mixin mechanism can simulate the same inheritance mechanisms as SmallTalk, BETA and CLOS. VanHilst[VN96] suggested a variation of mixin-based inheritance which enhances reusability of object-oriented programs.

C++ provides multiple inheritance, but does not linearize super classes. Therefore, it is impossible to do mixin-based inheritance with the C++'s multiple inheritance mechanism. A class which does not make a shared super class by multiple inheritance is sometimes called a mixin by C++ programmers, however, it should not be confused with the mixin used in this paper.

A mixin is largely similar to decorator pattern in Design Pattern[GRV95], with two differences: (1) a mixin does not allow modules to be exchanged during execution and (2) a mixin enables a new method to be added to a class, while a decorator can only have pre-fixed interface.

3.2.2 Program example using mixin

EPP's "SystemMixin plug-in" provides mixin-based inheritance [1]. The following describes the mixin

[1] Unlike intrinsic mixins, SystemMixin plug-ins actually provide programming by difference in the entire system consists of multiple classes, not in each class. However, this paper regards SystemMixins and intrinsic mixins as the same thing because EPP's parser consists of only one class.

```
class Foo {
  void m(char c){
    if (c == 'B') {
      doB();
    } else if (c == 'A') {
      doA();
    } else {
      doDefault();
    }
  }
}
```

Figure 2: A method definition which uses nested if statements.

features using the syntax extended by SystemMixin plug-in. First, Fig.2 shows a common method definition using nested `if` statements.

Next, Fig.3 shows the method definition divided into three mixins. Here, the method-invocation expression, `original`, introduced by SystemMixin plug-in, corresponds to the super method-invocation in traditional object-oriented languages.

Thus, mixins enable the user to divide a method, which used to be an inseparable unit, into multiple "method fragments"; and afterward the whole class can be constructed by combining multiple mixins.

```
SystemMixin Skeleton {
  class Foo {
    define void m(char c){
      doDefault();
    }
  }
}
SystemMixin A {
  class Foo {
    void m(char c){
      if (c == 'A') { doA(); }
      else { original(c); }
    }
  }
}
SystemMixin B {
  class Foo {
    void m(char c){
      if (c == 'B') { doB(); }
      else { original(c); }
    }
  }
}
```

Figure 3: A method definition by mixins.

3.2.3 Mixin implementation

EPP's SystemMixin plug-in implements mixins by translating all the method fragments incorporated in mixins into small Java classes. The method-invocation expression searches the hashtable of the receiver object for the object to implement a method segment and is translated into an expression to invocate its nested `call`. The problem with this implementation is that, with low efficiency, it is impossible to treat the mixin-defined class and the intrinsic Java class equally. However, this implementation is adopted because it allows separate compilation of the mixins.

C++ can also implement mixin-based inheritance by parameterizing the super class using the template mechanism. Fig.4 shows mixins defined by the template. Combining Skeletons A and B provides the class B¡A¡Skeleton¿¿. Note that mixins defined by the template do not allow separate compilation.

```
class Skeleton {
public:
  void m(char c){ doDefault(); }
};
template<class Super>
class A : public Super {
public:
  void m(char c){
    if (c == 'A') { doA(); }
    else { Super::m(c); }
  }
};
template<class Super>
class B : public Super {
public:
  void m(char c){
    if (c == 'B') { doB(); }
    else { Super::m(c); }
  }
};
```

Figure 4: Mixins defined by the template mechanism of C++.

3.3 Mixins Composing EPP

The EPP's parser is defined as the only class named `Epp`, with the class definition divided into multiple

mixins. Starting EPP combines all the mixins composing the standard Java parser and mixins composing the plug-in specified at the top of the source code to construct one parser (the class named `Epp`). EPP then generates the class instance and invokes the starting method to begin processing the input source code.

4 Implementation of extensible parser

4.1 Representation of tokens

One problem about extensible lexical analyzer is how the programmer extends the definition of data type returned by the lexical analyzer. Two possible solutions are as follows:

To provide the means of extending the token data type definition along with the means of extending the lexical analyzer.

To provide a general purpose data type in advance for all possible tokens.

Possibility (1) will require modification and recompilation of the source code of the parser because the lexical analyzer extension changes the data type, affecting the parser which processes the data type. Therefore, EPP implements possibility (2). The EPP's token data types, constructed so as to handle a wide range of extension, apply to almost any language extension without modifying data types. More specifically, all the tokens are expressed either in *literal* data types or *symbol* data types.

A literal is composed of a tag representing the kind of the literal and a string representing the content of the literal. For instance, an integer literal 123 is expressed as a tag `int` and a string `123`. In this way, a literal which was not included in the original syntax can be expressed with a new tag assigned.

All the tokens except literals-identifiers, key words such as `if` and `while`, operators, and special characters such as semicolons and parentheses-are expressed as symbols. EPP does not distinguish between keywords and identifiers; therefore, a new syntax can easily be added by simply extending the parser without modifying the lexical analyzer at all.

4.2 Recursive Descent Parser

This chapter describes the conventional implementation of the method of parsing non-terminals without using mixins. (Chapter 4.3 describes the implementation split into mixins.)

The following is an example of the production with alternatives of a prefix operator, parentheses, a right associative binary operator, a left associative binary operator, and a postfix operator [2].

```
Exp • •++ Exp | ( Exp ) | Term += Exp | Term
        | Exp + Term | Exp ++
```

Rewriting this production provides the form that can be parsed by the recursive descent parser: LL(1) grammar. (See the appendix for the details.) The recursive descent parser[ASU87] consists of functions which parse corresponding non-terminal and return the parsed abstract syntax trees.

Fig.5 shows a part of a recursive descent parser (without mixins) for the non-terminal `Exp` in the example production, where the three methods return abstract syntax trees as follows:

- `expTop` parses alternatives that are neither right recursion nor left recursion. For example, a prefix operator or parentheses.
- `expRight` parses alternative of right associative operators.
- `expLeft` parses alternatives that are left recursion. For example, postfix operators or left associative operators.

The roles of methods invoked from the program are as follows:

- `lookahead` returns the token currently being noticed.
- `match` reports an error if the current token differs from the argument value. Otherwise, it discards the current token and reads the next token.
- `matchAny` discards the current token unconditionally.

The program generates the abstract syntax trees as is expected from the production. For example, a += b += c generates (+= a (+= b c)); a + b + c generates (+ (+ a b) c).

4.3 Extensible Recursive Descent Parser

[2] In fact, in general language grammar, a non-terminal never mingles alternatives of parentheses and a binary operator; or a right associative binary operator and a left associative binary operator. Such grammar cannot generate expressions like `a + (b)` or `a + b += c`, and then, the grammar conflicts with human intuition.

```
Tree exp() {
  Tree tree = expTop();
  while (true){
    Tree newTree = expLeft(tree);
    if (newTree == null) break;
    tree = newTree;
  }
  return tree;
}
Tree expTop() {
  if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"preInc", exp());
  } else if (lookahead() == :"("){
    matchAny();
    Tree e = exp();
    match(:")");
    return new Tree(:"paren", e);
  } else {
    return expRight(exp1());
  }
}
Tree expRight(Tree tree) {
  if (lookahead() == :"+="){
    matchAny();
    return new Tree(:"+=",tree,exp());
  } else {
    return tree;
  }
}
Tree expLeft(Tree tree) {
  if (lookahead() == :"+"){
    matchAny();
    return new Tree(:"+",tree,exp1());
  } else if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"postInc", tree);
  } else {
    return null;
  }
}
Tree exp1() { return term(); }
```

Figure 5: A part of a recursive descent parser.

```
SystemMixin Exp {
  class Epp {
    define Tree exp(){
      Tree tree = expTop();
      while (true){
        Tree newTree = expLeft(tree);
        if (newTree == null) break;
        tree = newTree;
      }
      return tree;
    }
    define Tree expTop(){
      return expRight(exp1()); }
    define Tree expRight(Tree tree){
      return tree; }
    define Tree expLeft(Tree tree){
      return null; }
    define Tree exp1(){
      return term(); }
  }
}
```

Figure 6: A skeleton of a extensible parser.

```
SystemMixin Plus {
  class Epp {
    Tree expLeft(Tree tree) {
      if (lookahead() == :"+") {
        matchAny();
        return new Tree(:"+",tree,exp1());
      } else {
        return original(tree);
      }
    }
  }
}
```

Figure 7: A mixin which defines a left associative binary operator.

Splitting the program shown in Fig.5 into mixins makes it more modular and extensible. Removing *if-then* clauses and leaving *else* clauses in Fig.5 provides a skeleton as shown in Fig.6. `Exp` defined by the mixin is a method of parsing the following production:

```
Exp • •Term
```

New alternatives can be added to non-terminals by extending the methods expTop, expRight and expLeft in this program using mixins. Fig.7 shows a mixin which defines a left associative binary operator.

EPP defines dozens of kinds of non-terminals as a set of the mixins which define the skeleton as shown in Fig.6 and the mixins which add alternatives as shown in Fig.7. With a macro facilitating these definitions, the mixins in Fig.6 can be defined by the following one line:

```
defineNonTerminal(exp, term());
```

Also, the mixins which add the left associative binary operator in Fig. 7 can be defined by the following one line:

```
defineBinaryOperator(Plus, :"+", exp);
```

4.4 Lexical Analyzer Extension by Difference

A recursive descent lexical analyzer is extensible by difference. The EPP's lexical analyzer mainly consists of the following methods, whose behavior can be extended by mixins.

```
readToken
readId
readNumber
readOperator
readStringLiteral
readCharLiteral
readTraditionalComment
readEndOfLineComment
```

For example, Fig.8 shows the mixin that does not regard // as a beginning of a comment if it is followed by :. (This feature preserves extensibility and compatibility with Java. This is an example of a program using the mixin.

```
//: assert(predicate);
```

This line is simply regarded as a comment by the standard Java compiler, but works as an assert macro if the file is processed by EPP.)

```
SystemMixin CommentPragma {
  class Epp {
    Token readEndOfLineComment
                (EppInputStream in){
      if (in.peekc() == ':'){
        in.getc();
        return readToken(in);
      } else {
        return original(in);
      }
    }
  }
}
```

Figure 8: A mixin which extends the lexical analyzer.

4.5 Parser Module Deletion and Redefinition

EPP also provides a means of extending grammar other than programming by difference.

Grammar extension by programming by difference is somewhat limited in that (1) only new parser module addition is possible; current module deletion is impossible and (2) extension works only with prepared "hooks" (methods).

One way to perform extension without programming by difference is not to execute the original method invocation in adding mixins: i.e., to disregard the `original` method. For example, redefining exp1 in Fig.6 modifies the precedence of operators.

In addition to this, plug-ins have a mean of removing some parser definition modules (mixin) when plug-ins are loaded. By the mean, the grammar can be arbitrary modified.

The drawback of realizing the plug-ins through the above two means is that they are much less composable than extension only by programming by difference. A plug-in programmer/user has to trade-off extensibility against composability.

5 Ad-hoc Processes

5.1 Backtrack

EPP provides explicit backtrack with the lexical analyzer and the parser. Fig.9 shows a mixin which defines a new token "**" .

The argument EppInputStream is an input stream that backtracks at an arbitrary length by having the whole input file as a character array on the memory.

```
SystemMixin NewOp {
  class Epp {
    Token readOperator(EppInputStream in){
      if (in.peekc() == '*') {
        int p = in.pointer();
        in.getc();
        if (in.getc() == '*') {
           return :"**";
        }
        in.backtrack(p);
        return original(in);
      } else {
        return original(in);
      }
    }
  }
}
```

Figure 9: A mixin which defines a new token.

5.2 Context Sensitivity

A recursive descent parser easily implements context-sensitive processes in the lexical analyzer and the parser: the user just has to input the context information into the global variables (static variables, in Java terminology).

5.3 Error Recovery

Error recovery during parsing is easily implemented by Java's exception handling feature. The error handler just has to skip tokens till a particular token appears.

5.4 Line Number Managing

It is desirable to have information on "the line number at which the syntax began" in the Tree that is generated by parsing. The information helps generate clear error messages when errors occur during semantic analysis afterwards. EPP also uses the information to output each line of the source code at the same line after translation.

Lisp and Java easily implement line number managing. Lisp uses variables with dynamic scope, and Java uses static variables, stacks and try-finally syntax.

Fig.10 shows an example definition of the method exp with Java. All the non-terminal methods are defined in the same way. The Tree constructor obtains the line number at which it began by checking

```
Tree exp(){
  LineNumber.stack
    .push(currentLineNumber());
  try {
    ... Same as Fig.6 ...
  } finally {
    LineNumber.stack.pop();
  }
}
```

Figure 10: Managing line number information.

the top of the stack of the static variable LineNumber.stack.

6 Evaluation

6.1 Java Grammar Description

EPP incorporates a complete Java parser for JDK1.1 implemented as described in this paper. The Java grammar definition part consists of 105 mixins, 29 of which define the skeleton of non-terminals as shown in Fig.6.

Explicit backtrack was executed at the following points during implementation:

1. Distinction between constructor and method or field.
2. Distinction between static method/field and static initializer.
3. Distinction between method and field.
4. Distinction between local variable declaration and statement.

Some of the above can be parsed with LL(1) by rewriting grammar, but that lowers extensibility and modularity. Therefore, backtrack was adopted.

Field access syntax and cast syntax are implemented with lower modularity; it is impossible to make these elements highly modular unless type information is obtained during parsing.

6.2 Efficiency

The author tested the speed of EPP for processing of source code of EPP itself, consisting of 7218 lines, and obtained the following results.

- MMX Pentium 233MHz, Windows95, Microsoft SDK2.01: approximately 30 seconds

- UltraSPARC 200MHz, Solaris2.5.1, JDK1.1.3: approximately 40 seconds

The process consists of three parts: source code parsing, macro expansion of extended syntax, and source code emission after translation. The most time-consuming part is parsing. That is no problem for practical use, but a parser should work much faster. Three factors cause EPP's low speed:

1. Java interpreter overhead,
2. Mixin method invocation overhead, and
3. intrinsic low speed of parser description methods proposed by this paper (e.g., sequential search of alternatives with if-then-else and unnecessary invocation of method having no content).

The overhead of the current mixin method invocation is approximately 10 times slower than that of the standard Java class invocation. This is because EPP implements a mixin-defined fragment with a small Java object, invoking the method by searching the hashtable during execution. Mixin implementation speed should be improved in the future.

The overhead of sequentially searching for alternatives with if-then-else takes up much time. In order to improve that, one possible solution is to implement an optimized translator that is specialized to the EPP's parser source code. For example, if-then-else should be translated into a table search.

In general, having too many backtracks reduces parsing efficiency, but Java grammar never causes backtracks that will seriously reduce the speed, as proven by the following experiment. When the source code of Java.uil.Vector in JDK1.1.1 was parsed by EPP, 1214 tokens were appeared. There were 86 backtracks which caused 172 times extra invocation of `readTokens` method. The result shows that the invocation of `readToken` increases approximately 14

6.3 Ease of Debugging

Standard declarative description parser generators detect collisions and ambiguity of grammar to warn the user. Unfortunately, EPP's parser does not have this feature. Since it is intended for extending the grammar of already completed languages by difference, the parser is not intended as a tool for designing a new language grammar.

Nevertheless, it would be possible to construct a parser generator which generates mixins described in this paper.

Local debugging of a recursive descent parser is easy; the standard Java debugger and print statements work as in common programs.

7 Related work

ANTLR[ANT] and JavaCC[Jav] are recent top-down parser generators based on LL(k). According to the creators of these tools, the advantages of a top-down parser include ease of debugging and passing attribute values downward or upward during parsing. A bottom-up parser like LALR(1) does not have such features. Also, ANTLR can extend the existing grammar by difference through inheritance. JavaCC enables direct writing in part of the production, making writing easy with declarative description.

MPC++[Ish94], OpenC++[Chi95], JTRANS[KK97] and OpenJava[Tat] are extensible systems which can introduce new language features by providing compile time MOP (Meta object Protocol[KdRB91]) during compilation. Like EPP, their task is to perform complicated translation on an abstract syntax tree after parsing. Also, the grammar is extensible in a limited range. For example, MPC++ allows addition of new operators and statements.

Eli[GHL+92] is a compiler-generator which modularizes the grammar definition. It automatically generates a language processor using grammar and semantics definitions based on attribute grammar, and defines a new language by a kind of inheritance using existing definition modules.

Many "extensible languages" for grammar modification have been created, and most of them, including Lisp and C macros, define new grammar extension in the *on-the-fly* style, i.e. in the program to be processed. The problems with on-the-fly extension are that (1) it does not allow pre-compilation of the grammar extension code and therefore lacks efficiency, and (2) modifying or extending "the syntax to define syntax extension" itself often causes confusion. EPP has no such problems because it does not work in the on-the-fly style.

Camlp4[Rau] is an Objective Campl pre-processor whose grammar can be extended by adding modules. The extension can be done by difference with declarative description, and the modules can be compiled separately.

8 Conclusion

A method of constructing a highly modular and extensible parser by splitting the recursive descent

parser into small mixins was described. The syntax accepted by the parser can be extended with high composability over a wide range by adding mixins implemented by programming by difference. Also, in principle, removing existing mixins arbitrarily modifies grammar.

acknowledgment

The author wishes to thank Makoto Matsushita of Osaka University for his discussion about the method of describing an extensible parser in the early stages of the study, and Yves Rouder who was a visiting researcher of the Electrotechnical Laboratory for his helpful suggestions on EPP.

References

[ANT] Antlr home page. "http://www.ANTLR.org/".

[ASU87] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1987.

[BC90] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of E-COOP/OOPSLA'90*, 1990.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30(10) of *ACM Sigplan Notices*, pages 285 299, Austin, Texas, October 1995. ACM Press. "http://www.softlab.is.tsukuba.ac.jp /~chiba/openc++.html".

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121 131, February 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GRV95] Helm Gamma, E., R. R., Johnson, and J. Vlissides. *Design Patterns*. Addison-Welsley, 1995.

[Ich] Y. Ichisugi. EPP home page. "http://www.etl.go.jp/~epp".

[IR97] Y. Ichisugi and Yves Roudier. The extensible java preprocessor kit and a tiny data-parallel java. In *ISCOPE'97, California*, LNCS 1343, pages 153 160, December 1997.

[Ish94] Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994. "http://www.rwcp.or.jp/lab/mpslab/mpc++ /mpc++.html".

[Jav] Javacc home page. "http://www.suntest.com/JavaCC/".

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, 1991.

[KK97] A. Kumeta and M. Komuro. Meta - Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'97, Japan Society of Software Science and Technology*, March 1997.

[Rau] D. Rauglaudre. Camlp4 home page. "http://pauillac.inria.fr/camlp4/".

[Ste90] G.L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.

[Tat] M. Tatsubori. Open java home page. "http:// www.softlab.is.tsukuba.ac.jp /~mich/openjava/".

[VN96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *OOSPLA'96*, October 1996.

A Production Rewrite

The production defined as follow is rewritten so that it can be parsed by the recursive descent parser.

```
Exp • •++ Exp | ( Exp ) | Term += Exp | Term
      | Exp + Term | Exp ++
```

Split the production into two by introducing `ExpTop`.

```
Exp • •ExpTop | Exp + Term | Exp ++
ExpTop • •++ Exp | ( Exp ) | Term += Exp | Term
```

Remove the left recursion of `Exp` by introducing `ExpLoop` and rewrite `ExpTop` by introducing `ExpRight`.

```
Exp • •ExpTop ExpLoop
ExpTop • •++ Exp | ( Exp ) | Term ExpRight
ExpRight • •+= Exp | • •
ExpLoop • •+ Term ExpLoop | ++ ExpLoop | • •
```

Now this grammar can be parsed by recursive descent parser. Furthermore, by introducing `ExpLeft`, ExpLoop can be rewritten as follows:

```
Exp • •ExpTop ExpLoop
ExpTop • •++ Exp | ( Exp ) | Term ExpRight
ExpRight • •+= Exp | • •
ExpLoop • •ExpLeft ExpLoop | • •
ExpLeft • •+ Term | ++
```

Fig.5 shows a part of a recursive descent parser for non-terminals based on the above productions. Note that the tail recursion of ExpLoop is rewritten into Loop and embedded in the method `exp`; the method expLeft expresses returns a special value, null, if no matching alternatives.

Both the original and rewritten grammars are ambiguous. For example, the expression `++ a + 1` can be interpreted as either `(++ (+ a 1))` or `(+(++ a )1)` . The program in Fig. 5 parses this as `(+(++ a )1)`.

Index

EPP 1.1.0 Architecture Overview

Required Knowledge

In order to understand EPP documents, you will need basic knowledge of the Java lanaguage as well as basic knowledge of compilers. Specifically, you must understand the following words.

```
pass
recursive descent parser
token
literal
AST (Abstract Syntax Tree)
non-terminal
bootstrap
```

However, advanced expert knowledge should not be required.

Furthermore, the following concepts familiar to lisp programmers appear.

macro, immutable object, symbol, S-expression, backquote macro, dynamic variable

I shall explain these concepts as much as I can, so that readers who are not experienced in lisp can understand them.

EPP Description Language, Ld-2

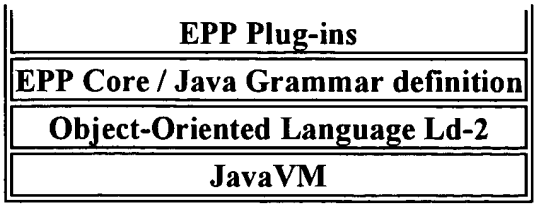
The source code of EPP is written using Java language that is extended with EPP. Specifically, the following five plug-ins are used.

- Symbol
- SystemMixin
- BackQuote
- AutoSplitFiles
- EppMacros

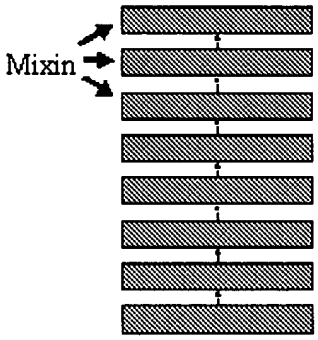
The source code for these five plug-ins are also written using these five plug-ins. Other than this, we have the Common Lisp source code for the EPP main portion, and source codes for five plug-ins. Using these to bootstrap, EPP can work even if you only have Java.

Of the five plug-ins, the SystemMixin plug-in is especially important. The SystemMixin plug-in implements a new object oriented language called Ld-2 on top of Java.

(Application Programs)



The Ld-2 lanaguage has a special inheritance mechanism that is different from that of the Java language. With this mechanism, a single class can be divided into multiple "components" called "mixins" which can be described separately. A class is built up by merging the multiple mixins.



In the current implementation, Ld-2 classes created by merging mixins are not compatible with classes of the Java language, and are defined using a different syntax. The syntax for calling methods is also different.

EPP Main Routine

When EPP is invoked, a program named EPP main routine executes. The program is a Java class with the following name.

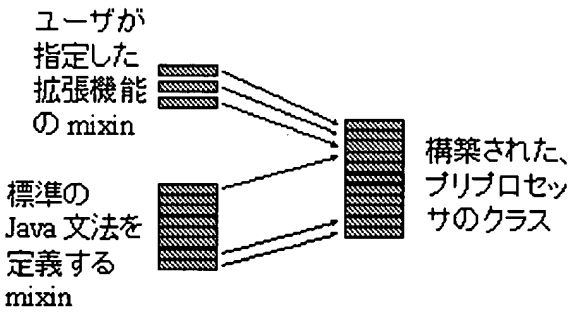
```
jp.go.etl.epp.epp.Epp
```

The EPP main routine builds an EPP preprocessor with different configurations for each file to be processed, and invokes those preprocessors.

An EPP preprocessor is an Ld-2 class created by merging multiple mixins. A preprocessor for a specific file is built by creating an Ld-2 class combining the mixin for the plug-in that was specified at the beginning of the file, and the "mixin that defines the standard preprocessor".

Plug-ins can only extend the behavior of EPP preprocessors. They cannot extend the behavior of the EPP main routine. However, by creating a subclass of the jp.go.etl.epp.epp.Epp Java class, you can create a customized EPP main routine.

EPP executes the translation process on a file-by-file basis by default. However, if you specify the -global option, EPP goes into Global Processing Mode and will process all files globally.



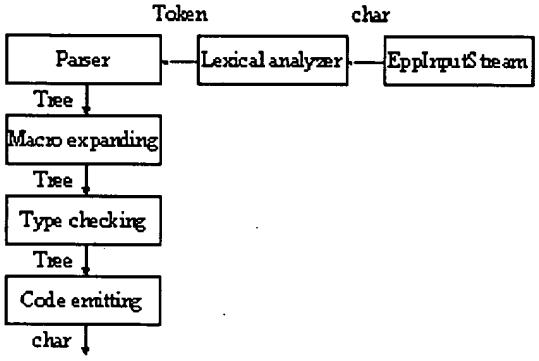
EPP Preprocessor

When the EPP preprocessor is invoked from the EPP main routine, the initialization method is called. After that, the input file is processed through the following four passes: parsing pass, macro expanssion pass, type checking pass, and code emitting pass.

The parsing pass will call the lexical analyzer as required. The lexical analyzer will read character-by-character from EppInputStream, a class that is the input stream for the EPP.

Plug-ins can extend the EppInputStream, lexical analyzer, parsing pass, macro expanssion pass, type checking pass, and code emitting pass.

You can also add additional passes prior to, and after the four passes. For further information on adding passes, refer to EPP Preprocessor Core.



Data Structure

Within the preprocessor, three data structures that describe the token, abstract syntax tree, and type are particularly important.

Tokens are data types that are returned by the lexical analyzer.

The abstract syntax tree is created by the parsing pass, and then translated by the type checking pass, passed to the code emitting pass for conversion to character strings and finally written to the output file.

The abstract syntax tree has nodes that have type information and those that do not. The type checking pass converts an abstract syntax tree without type information into an abstract syntax tree with type information.

The three data types mentioned above are all immutable objects. That is, you cannot modify their internal state from within a program.

Plug-ins cannot define subclasses of a class that describes tokens or abstract syntax trees. The data structure of tokens and abstract syntax trees are very versatile and new tokens and syntaxes can be expressed without adding new subclasses.

The Principle of the Extendable Parser

The parser is basically written in recursive descend style. You can add a mixin and extend the method of the parser in order to add a new alternative to the non-terminal. By backtracking and proceeding with context sensitive processing, the parser can handle non LL(1) type syntaxes. It can also handle left recursive rules.

For further information please refer to the following paper:
"□,,¢ƒ,ƒWƒ…ƒ‰ƒŠƒeƒB‚ÆŠg'£□«‚ðŽ□‚Â□\•¶‰ð□ÍŠí"

Related Information

For further information regarding writing plug-ins, please refer to the following.
Error Handling
Dynamic Variables

The Role of Each Source File

The source code for the EPP main portion is located under epp/src/level0/epp of the distributio package. The major fuctions of the files are shown below.

```
Epp.java              EPP main routine definition
EppCore.java          EPP preprocessor main portion
EppInputStream.java   EPP input stream
Lex.java              lexical analyzer
CompUnit.java         Java program top level syntax definition
TypeDecl.java         class, interface, method, field syntax defintion
Statement.java        statement syntax definition
ExpNonTerm.java       non-terminal definition related to expressions
```

Exp.java	expression syntax definition
TypeSystem.java	definition of the Type class and definition of type semantic
TypeCheck.java	definition of the standard Java type checking object
FileSig.java	definition of class types and seperate compilation processin

Index

Web Results **1** - **10** of about **113,000** for **JAVA preprocessor 1999**. (**0.23** seconds)

SOS : **Java Preprocessor**!!!!
SOS : **Java Preprocessor**!!!! dtv48@canal-plus.fr dtv48@canal-plus.fr Thu, 18 Nov **1999** 18:30:42 +0100. Previous message: SOS : **Java Preprocessor**! ...
https://lists.xcf.berkeley.edu/ lists/advanced-**java**/**1999**-November/003582.html - 5k - Cached - Similar pages

SOS : **Java Preprocessor**!!!!
SOS : **Java Preprocessor**!!!! Yogesh Pandeya yogesh@indts.com Thu, 18 Nov **1999** 22:31:39 -0000. Previous message: Rmiregistry; Next message: SOS : **Java** ...
https://lists.xcf.berkeley.edu/ lists/advanced-**java**/**1999**-November/003580.html - 4k - Jun 8, 2005 - Cached - Similar pages
[More results from https://lists.xcf.berkeley.edu]

EPP: An Extensible Pre-Processor Kit
1999-11-05 (EPP 1.1.0 beta 10) Error recovery in type-checking pass was implemented. ... "Extensible **Java Preprocessor** Kit and Tiny Data-Parallel **Java**" ...
staff.aist.go.jp/y-ichisugi/epp/ - 10k - Cached - Similar pages

Comp.compilers: Looking for a cpp(c **preprocessor**) written in **Java**
Looking for a cpp(c **preprocessor**) written in **Java** bharath3@my-dejanews.com (**1999**-05-03) ... I am looking for an implementation of c **preprocessor** in **java**. ...
compilers.iecc.com/comparch/article/99-05-013 - 3k - Cached - Similar pages

Visustin - C/C++, C#, **Java**, JSP, JavaScript and PHP flow charts
... support is built according to the ECMAScript standard (ECMA-262 3rd Edition **1999**). ... C/C++, C#, **Java**: **Preprocessor** directives are treated as comments. ...
www.aivosto.com/visustin/help/c-**java**.html - 8k - Cached - Similar pages

Java News from December, **1999**
Monday, December 27, **1999**. Sun's released version 1.2 of the **Java** ... ut wasn't the **preprocessor** one of the C obfuscations **Java** was supposed to save us from ...
www.cafeaulait.org/**1999**december.html - 25k - Cached - Similar pages

The Jonathan IDL2Java compiler
The compiler automatically launches jpp a simple **preprocessor** which implements a ... It maps individual IDL module scopes to **JAVA** packages and overrides the ...
jonathan.objectweb.org/current/doc/hrefs/idl2Java.html - 27k - Cached - Similar pages

Jfront - Operator Overloading for **Java**
The **Java** Grande working group has been working on this problem too. ... however we would rather see this **preprocessor** become a part of a mainstream ...
www.winternet.com/~gginc/jfront/ - 6k - Cached - Similar pages

The Jonathan IDL2Java compiler
The Jonathan idl2java Compiler generates the **java** classes and interfaces necessary ... The compiler automatically launches jpp a simple **preprocessor** which ...
pauillac.inria.fr/cdrom/ www/jonathan/doc/hrefs/idl2Java.html - 27k - Cached - Similar pages

JavaML: A Markup Language for **Java** Source, by Greg J. Badros

In this paper, I introduce the **Java** Markup Language, JavaML — an XML application for
... A framework for **preprocessor**-aware C source code analyses. ...
www9.org/w9cdrom/342/342.html - 80k - Cached - Similar pages

Gooooooooogle ▶
Result Page: 1 2 3 4 5 6 7 8 9 10 **Next**

JAVA preprocessor 1999 Search

Search within results | Language Tools | Search Tips | Dissatisfied? Help us improve

Google Home - Advertising Programs - Business Solutions - About Google

December 1999 Java News

Friday, December 31, 1999

Hot on the heels of version 2.2, Slava Pestov has released version 2.3pre1 of JEdit, the popular pure Java programmer's text editor. version 2.3 adds predefined abbreviation expansion, more toolbar buttcons, and assorted user interface improvements and bug fixes. JEdit is released under the the GNU General Public License.

Macintosh Runtime for Java 2.2 EA2 expires at midnight tonight. This isn't really a Y2K failure, just abunch of programmers underestimating how much time would be required to complete the next version. Starting tomorrow users will periodically see a dialog asking them to download a new version of the software.If you ignore the dialog, MRJ will still function correctly.

Steev Coco's released version 1.1 of Seymour, an open source Java IDE written in pure Java but mostly tested on the Macintosh.

IBM's released the second beta of the free Aglets Software Development Kit 1.1. Aglets are mobile agents written in Java. JDK 1.1 is required. JDK 1.2 will **not** work.

Sun's posted an early access release of the PersonalJava Runtime Environment for Windows CE 2.11/MIPS Platform on the Java Developer Connection (registration required). It allows you to run PersonalJava 1.1.1 applets and applications on devices running Windows CE 2.11 on a MIPS processor.

Thursday, December 30, 1999

Sun's posted a proposal for archiving/serializing Swing and JavaBeans based GUIs as XML. This may help fix the problem of inconsistent binary serialization formats between JDK versions that's especially troublesome for Swing applications. The proposal includes a sample Bean Builder tool for building GUI code from the proposed XML grammar. JDK 1.3 is required.

Wednesday, December 29, 1999

My essay on *Free Art? Free Software?* got picked up on Linux Today and other sites and mailing lists; and consequently I'm getting a lot of feedback, even in the relatively dead time between Christmas and New Year. I'm on vacation right now myself, so I can't respond to all of it immediately. I should get to it next week (barring the collapse of civilization as we know it). One of my New Year's resolutions is to get discussion forums working on this site; but in the meantime, Linux Today is hosting discussion of the article.

Monday, December 27, 1999

Sun's released version 1.2 of the Java Embedded Server, a small footprint application server for devices like cash registers and photocopiers. This is payware, but a 90-day free trial is available. Version 1.2 adds support for Java 2, Secure Sockets Layer (SSL), and the latest version of Swing, among other features.

Sun's released version 1.0.2 of the Java Advanced Imaging API for Windows and Sparc Solaris.

Version 1.1.3 of the Java 3D API OpenGL based implementation for Sparc Solaris and Windows.

Sun's posted version 0.95 of the JavaPhone Specification and Documentation in Acrobat PDF format. (At the end of the millenium, don't you think it's about time Sun's engineers and spec writers learned to use HTML like everybody else? Scott McNealy allegedly banned PowerPoint from Sun a couple of years ago. Perhaps it's time he banned Word and FrameMaker as well.) There's also class library documentation in the Javadoc HTML format.

Sunday, December 26, 1999

I've published an essay about some problems I see with Richard Stallman's call for free documentation. I'd be interested in hearing people's reactions and thoughts. Mostly I codified these thoughts while listening to RMS talk at The Bazaar a couple of weeks ago, and then reading a couple of his own documentation efforts for emacs and gdb.

Friday, December 24, 1999

Hot on the heels of version 2.2, Slava Pestov has released version 2.2.1 of JEdit, the popular pure Java programmer's text editor, to fix three bugs. JEdit is released under the the GNU General Public License.

Wednesday, December 22, 1999

I'm away on vacation in New Orleans for the holidays. Updates are liable to a be a little sporadic here until the New Year.

The Apache Jakarta Project has released Tomcat 3.0, the official reference implementation of the Java Servlet API and Java Server Pages.

Monday, December 20, 1999

Version 2.2 final of Slava Pestov's JEdit pure Java programmer's text editor is now available. JEdit is published under the the GNU General Public License.

IBM's alphaWorks has released some Enterprise JavaBeans for its WebSphere Application Server including Company Components, Address Components, Currency Components, and Natural Calendar Components.

Saturday, December 18, 1999

Sun's posted release candidate 2 of the JDK 1.2.2 for Linux on the Java Developer Connection (registration required). This release fixes assorted bugs.

Sun's posted final releases of four specifications:

- Java Server Pages 1.1
- Java Servlet API 2.2
- Enterprise JavaBeans 1.1
- Java 2 Enterprise Edition 1.2

Sun's released version 1.2 of the Java Embedded Server, a small footprint server for use in vending machines, pay phones, copiers, and so forth. It's not clear what the price is. It probably requires a contract with and royalty to Sun for actually shipping a product. However, as 90-day evaluation version is available for free.

Sun's released the Sun BluePrints Design Guidelines for J2EE, an integrated set of documentation and examples that illustrates "best practices" for developing and deploying J2EE compatible solutions. These are supposed to give developers of e-commerce applications examples of component design and optimization, division of development labor, and allocation of technology resources.

Friday, December 17, 1999

Neil Taylor's released the first beta of version 1.2 of Jake, a visual front end to javac, javap, and other Sun command line tools.

Beta 4-1 of Colt 1.0 is now available. Colt The Colt distribution provides an open source infrastructure for scalable scientific and technical computing in Java that contains data structures and algorithms for offline and online data analysis, linear algebra, multi-dimensional arrays, statistics, histogramming, Monte Carlo simulation, parallel and concurrent programming.

Version 0.1.81 of Tritonus, a GPL'd implementation of the JavaSound API for X86 Linux, is now available. This release can write .au files, adds clips for esd, and fixes assorted bugs.

Thursday, December 16, 1999

Sun's released the first beta of version 2.0 of the server-optimized HotSpot virtual machine for Windows. This beta requires JDK 1.2.2. Sun claims a 30% performance improvement over version 1.0.

Sun's also posted version 1.0 of the Java Transaction Service (JTS) specification. According to Sun,

> JTS specifies the implementation of a Transaction Manager which supports the JavaTM Transaction API (JTA) 1.0 Specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level. JTS uses the standard CORBA ORB/TS interfaces and Internet Inter-ORB Protocol (IIOP) for transaction context propagation between JTS Transaction Managers.
>
> A JTS Transaction Manager provides transaction services to the parties involved in distributed transactions: the application server, the resource manager, the standalone transactional application, and the Communication Resource Manager (CRM).

Tuesday, December 14, 1999

Sun's posted the first public review draft of version 1.2 of the PersonalJava specification. Comments are due by January 7, 2000.

Sun's also released a beta of Java Access Bridge for the Microsoft Windows 1.0. This product allows Windows based Assistive Technology to get at and interact with the Java Accessibility API. This release adds an installer, fixes some bugs, and adds a couple of things to the API. The installer only really works under Windows NT.

IBM's alphaWorks has released a new version of the Bean Scripting Framework that adds support for VBScript/JScript/PerlScript on Win32 platforms. The Bean Scripting Framework is an architecture for incorporating scripting into Java applications and applets.

Monday, December 13, 1999

Romain Guy's released version 2.6 of his Jext programmer's editor with many new features and bug fixes. Jext is written in pure Java.

No IDS Presume not relevant CK 6/10/05

Saturday, December 11, 1999

IBM's alphaWorks has released the first iteration of `CPreProcessorStream`, an `InputStream` for people writing parsers that need C/C++ preprocessing; that is, `#include`, `#if`, `#define` etc. Correct me if I'm wrong, ut wasn't the preprocessor one of the C obfuscations Java was supposed to save us from? About the only plausible use I can imagine for this would be if you were writing a C or C++ compiler in Java, and why would you want to do that?

Friday, December 10, 1999

I'm afraid I missed most of the Java Business Expo due to a brief bout with the flu. By the time I was feeling better, all I had to time to do was see the Penn and Teller show (lots of fun) and make a brief pass through the show floor. The floor was noticeable for the large number of booths filled with engineers talking to customers about real products as opposed to flashy light shows and booth bunnies. I really wish I had more time to chat with more vendors.

The one thing I did had time to look at that looked worthy of further investigation was JVision, a $99 payware tool for reverse engineering Java code into UML diagrams. The Linux version is free, and all version are free for academic use. I've needed something like this for my books and courses for a while now, but everything I've seen up till now cost four figures or more (a price point where I just rule the product out of consideration.) It's available on Linux, NT, and Solaris. There's a 30-day demo available. I'll try it out and let you know what I think.

IBM's alphaWorks has posted a new copy of their IRC Client for Java just to extend the expiration date to June 30, 2000. (Wouldn't it be easier just to release one that doesn't expire?)

Thursday, December 9, 1999

Borland's released JBuilder Foundation for Linux, Solaris, and Windows. This is a free beer, pure Java IDE based on Java 2.

Wednesday, December 8, 1999

Sun has officially withdrawn Java from the ECMA standardization process because the ECMA refused to be used as a rubber stamp for whatever Sun chose to submit and insisted on retaining control of the standard. Earlier, Sun pulled out of the ISO process for the same reason. Sun is adamant in its refusal to allow anyone but Sun to have any authority or control over Java. They are willing to listen to companies or individuals who have good ideas (as long as they aren't Microsoft or Bill Gates, in which case Sun won't even listen) but it simply refuses to put itself in a position where anybody else can make it put something in or take something out of Java against Sun's expressed wishes. This effectively ends any hope of a Sun-supported, de jure Java standard. The ECMA is debating whether or not to go forward without Sun's participation.

One of the main purposes of de jure standardization like that performe

SOS : Java Preprocessor!!!!

dtv48@canal-plus.fr dtv48@canal-plus.fr
Thu, 18 Nov 1999 18:30:42 +0100

- Previous message: SOS : Java Preprocessor!!!!
- Next message: misaligned ImageIcons when seen over http
- **Messages sorted by:** [date] [thread] [subject] [author]

---

try Kiev at this adress
http://www.forestro.com/kiev/home.html

SeB

-----Message d'origine-----
De : yogesh@indts.com [mailto:yogesh@indts.com]
Envoyé : jeudi 18 novembre 1999 23:32
À : Michael.Heist@unilever.com
Cc : advanced-java@xcf.berkeley.edu
Objet : RE: SOS : Java Preprocessor!!!!

Yeah I need it written in java so that i can bundle it and do the processing
at install time....
Do you know any??/
Thanks

-----Original Message-----
From: Michael Heist [SMTP:Michael.Heist@unilever.com]
Sent: Thursday, November 18, 1999 7:56 PM
To: yogesh@indts.com
Cc: advanced-java@xcf.berkeley.edu
Subject: Re: SOS : Java Preprocessor!!!!

does the utility have to be written in java? most c compilers have a flag
for preprocess only
so you could precompile your java files with your c compiler. for some
reason this appeals to
me greatly.

#undefine GOSLING

>Hi List,
>I need it urgently!!
>I need a utility in Java which does pre processing like C or C++. I want to
give #ifdef and
>#else #endif in my java code inside the class definition and preprocess it
to generate new java
>file appropriately.
>thanks,
>Yogesh

To unsubscribe, mail advanced-java-unsubscribe@xcf.berkeley.edu
To get help, mail advanced-java-help@xcf.berkeley.edu

To unsubscribe, mail advanced-java-unsubscribe@xcf.berkeley.edu
To get help, mail advanced-java-help@xcf.berkeley.edu

To unsubscribe, mail advanced-java-unsubscribe@xcf.berkeley.edu
To get help, mail advanced-java-help@xcf.berkeley.edu

- Previous message: SOS : Java Preprocessor!!!!
- Next message: misaligned ImageIcons when seen over http
- **Messages sorted by:** [date] [thread] [subject] [author]

SOS : Java Preprocessor!!!!

Frank D. Greco fgreco@crossroadstech.com
Thu, 18 Nov 1999 09:31:05 -0500

- Previous message: SOS : Java Preprocessor!!!!
- Next message: More offscreen JEditorPane rendering...
- **Messages sorted by:** [date] [thread] [subject] [author]

```
At 07:03 PM 11/18/99 +0000, Yogesh Pandeya wrote:
>Hi List,
>I need it urgently!!
>I need a utility in Java which does pre processing like C or C++. I want
>to give #ifdef and #else #endif in my java code inside the class
>definition and preprocess it to generate new java file appropriately.
>thanks,
>Yogesh

        errr... how about using the C/C++ preprocessor itself?  The C/C++
        preprocessor doesn't know anything about C/C++... its just a
        regular text preprocessor.  Or how about even using 'm4'?  Its
        been around for over 20 years...

        Frank G.
+=======================================================================+
| Crossroads Technologies Inc, 55 Broad Street, 28th Fl, NYC, NY 10004 |
| Email: fgreco@CrossroadsTech.com          Web: www.CrossroadsTech.com |
| Voice: 212-482-5280 x229                  Fax: 212-482-5281           |
+=======================================================================+


---
To unsubscribe, mail advanced-java-unsubscribe@xcf.berkeley.edu
To get help, mail advanced-java-help@xcf.berkeley.edu
```

- Previous message: SOS : Java Preprocessor!!!!
- Next message: More offscreen JEditorPane rendering...
- **Messages sorted by:** [date] [thread] [subject] [author]

Java Glossary : preprocessor


Last updated 2005-05-20 by Roedy Green : ©1996-2005 Canadian Mind Products : feedback

Java definitions: 0-9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

You are here : home : Java Glossary : P words : preprocessor.

preprocessor

Java does not have an official preprocessor. You can alway use C/C++'s, cpp. However, Java does guarantee to eliminate dead code inside an if like this:

```
static final boolean DEBUGGING = false;

if ( DEBUGGING )
   {
   System.out.println(i);
   }
```

The code inside the if still has to be valid. Unfortunately you can't use the technique, for example, to select one of two different import statements.

DBC PP ¤ EPP ¤ IPP ¤ JPP ¤ Scriptic

Canadian Mind Products

mindprod.com IP:[24.87.56.253]

Your face IP:[151.207.240.3]

You are visitor number 2143.

Please send errors, omissions and suggestions

to improve this page to **Roedy Green**.


home

You can get a fresh copy of this page from:

http://mindprod.com/jgloss/preprocessor.html

Volunte

The ad abo
Google.con

or possibly from your loca

J:\mindprod\jgloss\pr

Lencom's shareware pics

Lencom Software Inc.

Home

Java Preprocessor 0.1 prerelease

- **Bulk Email Marketing**
- **About Us**
- **Other software**
- **Bulletproof hosting**

Avi Mpeg Joiner Splitter

Other Downloads

Fast Email Sender
Easy send rich HTML messages having your Newsletters customized as You want using this software!
Download Now

Fast Newsgroups Extractor
Process hundreds of thousands newsgroups messages and download attachments, information , addresses and more...
Download Now

Publisher : GehtSoft

Java Preprocessor allows you to use the C-like macros in your Java sources. The preprocessor is the superstructure over the Java compiler and requires the JDK. The preprocessor translates your sources with the preprocessor macros into the Java sources and calls the Java compiler. The messages about errors are translated into Microsoft C-like messages. The positions of errors will point to the positions in original files, not to the resulting Java sources.

Download NOW

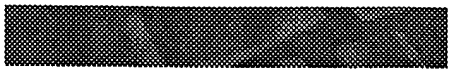
Fast Email E
Extracts emai
search engine
Download

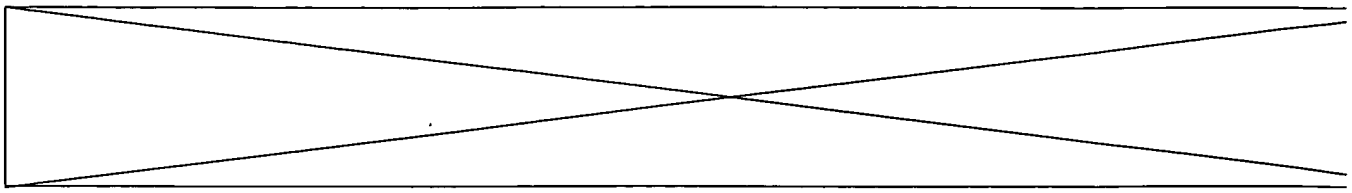
Advanced O
Extractor
Extracts any c
the big websit
Download

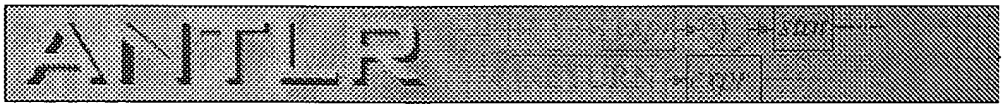
Fast Email V
Validates ema
without sendi
anything to us
Download

back next
Pages: 11926 , 11927 , 11928 , 11929 , 11930 , 11931 , 11932 , 11933 , 11934 , 11935 , 11936 , 11937 , 11938 , 11939 , 11940

Send comments, corrections, and suggestions to support@lencom...com







» Home
» Download
» News
»Using ANTLR
» Documentation
» FAQ
» Articles
» Grammars
» File Sharing
» Code API
» Tech Support
»About ANTLR
» What is ANTLR
» Why use ANTLR
» Showcase
» Testimonials
» Getting Started
» Software License
» ANTLR WebLogs
» ANTLR Workshops
»StringTemplate
»TML
»PCCTS
»Feedback
»Credits
»Contact

Preprocessor

antlr.preprocessor

Class Preprocessor

```
java.lang.Object
  └ antlr.Parser
      └ antlr.LLkParser
          └ antlr.preprocessor.Preprocessor
```

All Implemented Interfaces:
PreprocessorTokenTypes

public class **Preprocessor**
extends LLkParser
implements PreprocessorTokenTypes

Field Summary

static java.lang.String []	tokenNames
static BitSet	tokenSet_0
static BitSet	tokenSet_1
static BitSet	tokenSet_2
static BitSet	tokenSet_3
static BitSet	tokenSet_4
static BitSet	tokenSet_5
static BitSet	tokenSet_6

static BitSet	_tokenSet_7
static BitSet	_tokenSet_8

Fields inherited from interface antlr.preprocessor.PreprocessorTokenTypes

ACTION, ALT, ARG_ACTION, ASSIGN_RHS, BANG, CHAR_LITERAL, COMMA, COMMENT, CURLY_BLOCK_SCARF, DIGIT, ELEMENT, EOF, ESC, HEADER_ACTION, ID, ID_OR_KEYWORD, LITERAL_catch, LITERAL_class, LITERAL_exception, LITERAL_extends, LITERAL_private, LITERAL_protected, LITERAL_public, LITERAL_returns, LITERAL_throws, LITERAL_tokens, LPAREN, ML_COMMENT, NEWLINE, NULL_TREE_LOOKAHEAD, OPTIONS_START, RCURLY, RPAREN, RULE_BLOCK, SEMI, SL_COMMENT, STRING_LITERAL, SUBRULE_BLOCK, TOKENS_SPEC, WS, XDIGIT

Constructor Summary

Preprocessor(ParserSharedInputState state)
Preprocessor(TokenBuffer tokenBuf)
Preprocessor(TokenStream lexer)

Method Summary

antlr.preprocessor.Grammar	class_def(java.lang.String file, Hierarchy hier)
java.lang.String	exceptionGroup()
java.lang.String	exceptionHandler()
java.lang.String	exceptionSpec()
void	grammarFile(Hierarchy hier, java.lang.String file)
IndexedVector	optionSpec (antlr.preprocessor.Grammar gr)
void	reportError(RecognitionException e)

	Delegates the error message to the tool if any was registered via #initTool (antlr.Tool)
void	reportError(java.lang.String s) Delegates the error message to the tool if any was registered via #initTool (antlr.Tool)
void	reportWarning(java.lang.String s) Delegates the warning message to the tool if any was registered via #initTool (antlr.Tool)
void	rule(antlr.preprocessor.Grammar gr)
void	setTool(Tool tool) In order to make it so existing subclasses don't break, we won't require that the antlr.Tool instance be passed as a constructor element.
java.lang.String	superClass()
java.lang.String	throwsSpec()

Methods inherited from class antlr.LLkParser

consume, LA, LT, traceIn, traceOut

Methods inherited from class antlr.Parser

addMessageListener, addParserListener, addParserMatchListener, addParserTokenListener, addSemanticPredicateListener, addSyntacticPredicateListener, addTraceListener, consumeUntil, consumeUntil, getAST, getASTFactory, getFilename, getInputState, getTokenName, getTokenNames, getTokenTypeToASTClassMap, isDebugMode, mark, match, match, matchNot, panic, removeMessageListener, removeParserListener, removeParserMatchListener, removeParserTokenListener, removeSemanticPredicateListener, removeSyntacticPredicateListener, removeTraceListener, rewind, setASTFactory, setASTNodeClass, setASTNodeType, setDebugMode, setFilename, setIgnoreInvalidDebugCalls, setInputState, setTokenBuffer, traceIndent

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

_tokenNames

```
public static final java.lang.String[] _tokenNames
```

_tokenSet_0

```
public static final BitSet _tokenSet_0
```

_tokenSet_1

```
public static final BitSet _tokenSet_1
```

_tokenSet_2

```
public static final BitSet _tokenSet_2
```

_tokenSet_3

```
public static final BitSet _tokenSet_3
```

_tokenSet_4

```
public static final BitSet _tokenSet_4
```

_tokenSet_5

```
public static final BitSet _tokenSet_5
```

_tokenSet_6

```
public static final BitSet _tokenSet_6
```

_tokenSet_7

```
public static final BitSet _tokenSet_7
```

_tokenSet_8

```
public static final BitSet _tokenSet_8
```

Constructor Detail

Preprocessor

```
public Preprocessor(TokenBuffer tokenBuf)
```

Preprocessor

```
public Preprocessor(TokenStream lexer)
```

Preprocessor

```
public Preprocessor(ParserSharedInputState state)
```

Method Detail

setTool

```
public void setTool(Tool tool)
```

In order to make it so existing subclasses don't break, we won't require that the antlr.Tool instance be passed as a constructor element. Instead, the antlr.Tool instance should register itself via `#initTool(antlr.Tool)`

Throws:

`java.lang.IllegalStateException` - if a tool has already been registered

Since:

2.7.2

---

reportError

```
public void reportError(java.lang.String s)
```

Delegates the error message to the tool if any was registered via `#initTool`

(antlr.Tool)

Overrides:
reportError in class Parser
Since:
2.7.2

---

reportError

```
public void reportError(RecognitionException e)
```

Delegates the error message to the tool if any was registered via #initTool(antlr.Tool)

Overrides:
reportError in class Parser
Since:
2.7.2

---

reportWarning

```
public void reportWarning(java.lang.String s)
```

Delegates the warning message to the tool if any was registered via #initTool(antlr.Tool)

Overrides:
reportWarning in class Parser
Since:
2.7.2

---

grammarFile

```
public final void grammarFile(Hierarchy hier,
                              java.lang.String file)
                       throws RecognitionException,
                              TokenStreamException
```

Throws:
RecognitionException
TokenStreamException

---

optionSpec

```
public final IndexedVector optionSpec(antlr.preprocessor.Grammar gr)
                               throws RecognitionException,
                                      TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

---

class_def

```
public final antlr.preprocessor.Grammar class_def(java.lang.String file,
                                                   Hierarchy hier)
                                            throws RecognitionException,
                                                   TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

---

superClass

```
public final java.lang.String superClass()
                                 throws RecognitionException,
                                        TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

---

rule

```
public final void rule(antlr.preprocessor.Grammar gr)
            throws RecognitionException,
                   TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

---

throwsSpec

```
public final java.lang.String throwsSpec()
                                 throws RecognitionException,
                                        TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

---

exceptionGroup

```
public final java.lang.String exceptionGroup()
```

throws RecognitionException,
TokenStreamException

Throws:

RecognitionException
TokenStreamException

---

exceptionSpec

```
public final java.lang.String exceptionSpec()
                                    throws RecognitionException,
                                           TokenStreamException
```

Throws:

RecognitionException
TokenStreamException

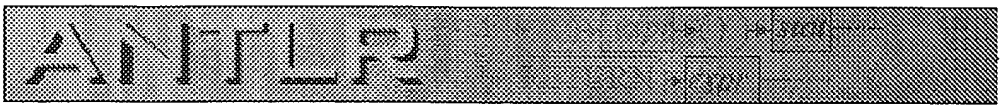
---

exceptionHandler

```
public final java.lang.String exceptionHandler()
                                    throws RecognitionException,
                                           TokenStreamException
```

Throws:

RecognitionException
TokenStreamException



ANTLR Reference Manual

ANTLR Reference Manual

Credits

Project Lead and Supreme Dictator
Terence Parr
University of San Franciso

Support from
jGuru.com
Your View of the Java Universe

Help with initial coding
John Lilly, Empathy Software

C++ code generator by
Peter Wells and *Ric Klaren*

C# code generation by
Micheal Jordan, Kunle Odutola and Anthony Oguntimehin.

Python's universe has been extended by
Wolfgang Häfelinger and *Marq Kole*

Infrastructure support from Perforce:
The world's best source code control system

Substantial intellectual effort donated by
Loring Craymer
Monty Zukowski
Jim Coker
Scott Stanchfield
John Mitchell
Chapman Flack (UNICODE, streams)

Source changes for Eclipse and NetBeans by
Marco van Meegen and Brian Smith

ANTLR Version 2.7.5
January 28, 2005

What's ANTLR

ANTLR, ANother Tool for Language Recognition, (formerly **PCCTS**) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C++, or C# actions [You can use PCCTS 1.xx to generate C-based parsers].

Computer language translation has become a common task. While compilers and tools for traditional computer languages (such as C or Java) are still being built, their number is dwarfed by the thousands of mini-languages for which recognizers and translators are being developed. Programmers construct translators for database formats, graphical data files (e.g., PostScript, AutoCAD), text processing files (e.g., HTML, SGML). ANTLR is designed to handle all of your translation tasks.

Terence Parr has been working on ANTLR since 1989 and, together with his colleagues, has made a number of fundamental contributions to parsing theory and language tool construction, leading to the resurgence of LL(k)-based recognition tools.

Here is a chronological history and credit list for ANTLR/PCCTS.

See ANTLR software rights.

Check out **Getting started** for a list of tutorials and get your questions answered at the ANTLR FAQ at jguru.com

See also http://www.ANTLR.org and **glossary**.

If you are looking for the previous main version (PCCTS 1.33) of ANTLR rather than the new Java-based version, see Getting started with PCCTS.

Download ANTLR.

ANTLR 2.7.5 release notes

ANTLR Meta-Language

- Meta-Language Vocabulary
- Header Section
- Parser Class Definitions
- Lexical Analyzer Class Definitions
- Tree-parser Class Definitions
- Options Section
- Tokens Section
- Grammar Inheritance
- Rule Definitions
- Atomic Production elements
- Simple Production elements

Lexical Analysis with ANTLR

ANTLR Tree Parsers

C++ Runtime model

C# Runtime model

Python Runtime model

ANTLR Tree Construction